

## APPENDIX I -HDL DESCRIPTION

This file has been redacted

--  
--  
--  
--  
--  
--  
--  
--  
--  
--

Confidential Information  
Limited Distribution to Authorized Persons Only  
Created 1996 and Protected as an Unpublished Work  
Under the U.S. Copyright Act of 1976.  
Copyright © 1996 - 2001 ARC CORES LTD.  
All Rights Reserved.

-- Inputs and Outputs:

--  
--  
--  
--

-- L indicates a latched signal, U indicates an signal produced by logic.

--

----- Stage 1 - Opcode fetch -----

--

-- in pliw[31:0] U The instruction word supplied by the memory controller.  
-- It is considered to be valid when the ivalid signal is  
-- true.

--

-- in ivalid U Qualifying signal for pliw[31:0]. When it is low, this  
-- indicates that the m/c has not been able to fetch the  
-- requested opcode, and that the program counter should not  
-- be incremented. The pipeline might be stalled, depending  
-- upon whether the instruction in stage 2 needs to look at  
-- the instruction in stage 1.

--

-- When it is true, the instruction is clocked into  
-- pipeline stage 2 provided that the pipeline is able to  
-- move on.

--

-- in ivic U Indicates that all values in the cache are to be  
-- invalidated. (it stands for InVAlidate Instruction Cache).  
-- It is anticipated that this signal will be generated from a  
-- decode of an SR instruction.

--

-- Note that due to the pipelined nature of the ARC, up to

three

--

-- instructions could be issued following the SR which

generates

--

-- the ivic signal. Cache invalidates must be suppressed when a  
-- line is being loaded from memory. This is done at the  
-- auxiliary register which generates ivic.

--

-- out pcen U Program counter enable. When this signal is true, the pc  
-- will change at the end of the cycle, indicating that the  
-- memory controller needs to do a fetch on the next cycle  
-- using the address which will appear on currentpc[], which  
-- is supplied from aux\_regs.vhd.

--

-- This signal is affected by interrupt logic and all the  
-- other pipeline stage enables.

--

-- out ifetch U This signal, similar to pcen, indicates to the memory  
-- controller that a new instruction is required, and should  
-- be fetched from memory from the address which will be  
-- clocked into currentpc[25:2] at the end of the cycle. It  
-- is also true for one cycle when the processor has been

--

```

-- started following a reset, in order to get the ball
-- rolling.
-- An instruction fetch will also be issued if the host
-- changes the program counter when the ARC is halted,
-- provided it is not directly after a reset.
-- The ifetch signal will never be set true whilst the
-- memory controller is in the process of doing an
-- instruction fetch, so it may be used by the memory
-- controller as an acknowledgement of instruction receipt.
--
-- out ipending      U This signal is true when an instruction fetch has been
--                   issued, and it has not yet completed. It is not true
--                   directly after a reset before the ARC has started, as no
--                   instruction fetch will have been issued. It is used to
--                   hold off host writes to the program counter when the ARC
--                   is halted, as these accesses will trigger an instruction
--                   fetch.
--
-- out plint         U indicates that an interrupt has been detected, and an
--                   interrupt-op will be inserted into stage 2 on the next
--                   cycle, (subject to pipeline enables) setting p2int true.
--                   This signal will have the effect of canceling the
--                   instruction currently being fetched by stage 1 by causing
--                   p2iv to be set false at the end of the cycle when plint
--                   is true.
--
-- out en1           U Stage 2 pipeline latch control. True when an instruction
--                   is being latched into pipeline stage 2. Will be true
--                   at different times to pcen, as it allows junk instructions
--                   to be latched into the pipeline.
--                   *** A feature of this signal is that it will allow an
--                   instruction be clocked into stage 2 even when stage 3
--                   is halted, provided that stage 2 contains a killed
--                   instruction (i.e. p2iv = '0'). This is called a
--                   'catch-up'. ***
--
----- Stage 2 - Operand fetch -----
--
-- out en2           U Pipeline stage 2 enable. When this signal is true, the
--                   instruction in stage 2 can pass into stage 3 at the end
--                   of the cycle. When it is false, it will hold up stage 2
--                   and stage 1 (pcen).
--
-- out p2i[4:0]      L Opcode word. This bus contains the instruction word
--                   which is being executed by stage 2. It must be qualified
--                   by p2iv.
--
-- out p2iv           L Opcode valid. This signal is used to indicate that the
--                   opcode in pipeline stage 2 is a valid instruction. The
--                   instruction may not be valid if a junk instruction has
--                   been allowed to come into the pipeline in order to allow
--                   the pipeline to continue running when an instruction
--                   cannot be fetched by the memory controller.
--
-- out fsla[5:0]      L Source 1 register address. This is the B field from the
--                   instruction word, sent to the core registers (via hostif)

```

```

--          and the LSU. It is qualified for LSU use by slen.
--
-- out s2a[5:0]  L Source 2 register address. This is the C field from the
--                instruction word, sent to the core registers and the
--                LSU. It is qualified for LSU use by s2en.
--
-- out dest[5:0] L Destination register address. This is the A field from
--                the instruction word, sent to the LSU for register
--                scoreboarding of loads. It is qualified by the desten
--                signal.
--
-- out slen      U This signal is used to indicate to the LSU that the
--                instruction in pipeline stage 2 will use the data from
--                the register specified by fsla[5:0]. If the signal is
--                not true, the LSU will ignore fsla[5:0]. This signal
--                includes p2iv as part of its decode.
--
-- out s2en      U This signal is used to indicate to the LSU that the
--                instruction in pipeline stage 2 will use the data from
--                the register specified by s2a[5:0]. If the signal is
--                not true, the LSU will ignore s2a[5:0]. This signal
--                includes p2iv as part of its decode.
--
-- in  xholdup12 U From extensions. This signal is used to hold up pipeline
--                stages 1 and 2 (pcen, en1 and en2) when extension logic
--                requires that stage 2 be held up. For example, a core
--                register is being used as a window into SRAM, and the SRAM
--                is not available on this cycle, as a write is taking place
--                from stage 4, the writeback stage. Hence stage 2 must be
held
--                to allow the write to complete before the load can happen.
--                Stages 3 and 4 will continue running.
--
-- out desten    U This signal is used to indicate to the LSU that the
--                instruction in pipeline stage 2 will use the data from
--                the register specified by dest[5:0]. If the signal is
--                not true, the LSU will ignore dest[5:0]. This signal
--                includes p2iv as part of its decode.
--
-- out p2offset[19:0] L This bus carries the region of the instruction which
--                contains the branch offset. It is used by the program
--                counter generation logic when the instruction in stage 2
--                is a Bcc/BLcc or LPcc.
--
-- out p2condtrue U This signal is produced from the result of the internal
--                stage 2 condition code unit or from an extension cc unit
--                (if implemented). A bit (bit 5) in the instruction selects
--                between the internal and extension cc unit results.
--                As stage 2 conditionals are only used by branch and jump
--                instructions, the logic to produce this signal is simpler
--                than that required from p3condtrue, which takes into
--                account the complications presented by short immediate
--                data registers, amongst other things. When using
--                p2condtrue, a decode for a branch/jump instruction must
--                always be included along with a check for p2iv = '1'.
--
-- out p2setflags L This is bit 8 from the instruction word at stage 2,

```

```

-- i.e. the .F or setflags bit used in the jump instruction.
-- It is used in flags.vhd to determine whether the flags
-- should be loaded by a jump instruction. The stage 3
-- signal p3setflags is much more complicated, having to
-- take into account the complications presented by short
-- immediate data, amongst other things.
--
-- in p2int      L This signal indicates that an interrupt jump instruction
--               (fantasy instruction) is currently in stage 2. This signal
--               has a number of consequences throughout the system,
--               causing the interrupt vector (int_vec[25:2]) to be put
--               into the PC, and causing the old PC to be placed into
--               the pipeline in order to be stored into the appropriate
--               interrupt link register.
--               Note that p2int and p2iv are mutually exclusive.
--
-- out p2jblcc   U True when a JLcc or BLcc instruction is in stage 2.
--               Does not include p2iv.
--               Used in conjunction with the branch delay slot mode which
--               is re-created from the short immediate field.
--
-- out p2st      U This signal is used by coreregs.vhd. It is produced from
--               a decode of p2i[4:0], p2iw(25) (check for SR) and
--               does not include p2iv.
--
-- out p2ldo     U True when p2i[4:0] = oldo, and p2iw(13) = '0', which
--               indicates that the instruction is an LDO, not an LR which
--               is an encoding of the LDO instruction.
--               This signal is used by coreregs.vhd to switch short imm
--               data onto a source bus when an LDO instruction is used.
--               Does not include p2iv.
--
-- out p2lr      U True when p2i[4:0] = oldo, and p2iw(13) = '1', which
--               indicates that the instruction is the auxiliary register
--               load instruction LR, not a memory load LDO instruction.
--               This signal is used by coreregs.vhd to switch the
--               currentpc bus onto the source2 bus (which is then passed
--               through the same logic as the interrupt link register)
--               in order to get the correct value of pc when it is read
--               by an LR instruction.
--               Does not include p2iv.
--
-- out mload2    U This signal indicates to the LSU that there is a valid
--               load instruction in stage 2. It is produced from a decode
--               of p2i[4:0], p2iw(13) (to exclude LR) and the p2iv signal.
--
-- out mstore2   U This signal indicates to the actionpoint mechanism when
--               selected that there is a valid store instruction in stage
--               2. It is produced from a decode of p2i[4:0], p2iw(13)
--               (to exclude SR) and the p2iv signal.
--
-- in holdup12   U From lsu.vhd. This signal is used to hold up pipeline
--               stages 1 and 2 (pcen and en2) when the load store unit
--               finds a register being used by the instruction at stage
--               2 which is the destination of a delayed load. It will
--               also be set when the scoreboard unit is full and the
--               ARC attempts to do another load. Stages 3 and 4 will

```

```

-- will continue running.
--
-- in aluflags[3:0] L ALU flags, direct from the latches in flags.vhd
--
-- in x_p2noscl U From extensions. Indicates that the register referenced
-- by fs1a[5:0] is not available for shortcutting. This signal
-- should only be set true when the register in question is
-- an extension core register. This signal is ignored unless
-- constant xt_corereg is set true.
--
-- in x_p2nosc2 U From extensions. Indicates that the register referenced
-- by s2a[5:0] is not available for shortcutting. This signal
-- should only be set true when the register in question is
-- an extension core register. This signal is ignored unless
-- constant xt_corereg is set true.
--
-- out dorel U True when a relative branch (not jump) is going to happen.
-- Relates to the instruction in p2. Includes p2iv.
--
-- out dojcc U True when a jump is going to happen.
-- Relates to the instruction in p2. Includes p2iv.
--
-- out p2killnext U True when the instruction in stage 2 is a branch/jump type
-- operation which will kill the following delay slot
-- instruction.
-- The following operation will be marked invalid when it is
-- passed from stage 1 into stage 2.
--
----- Stage 3 - ALU -----
--
-- out en3 U Pipeline stage 3 enable. When this signal is true, the
-- instruction in stage 3 can pass into stage 4 at the end
-- of the cycle. When it is false, it will probably hold up
-- stages one (pcen), two (en2), and three.
--
-- out p3i[4:0] L Opcode word. This bus contains the instruction word
-- which is being executed by stage 3. It must be
-- qualified by p3iv.
--
-- out p3a[5:0] L Instruction A field. This bus carries the region of
-- the instruction which contains the operand dest field.
--
-- out p3c[5:0] L Instruction C field. This bus carries the region of
-- the instruction which contains the operand C field. This
-- is used to encode extra single-operand functions onto
-- the FLAG instruction opcode.
--
-- out p3iv L Opcode valid. This signal is used to indicate that the
-- opcode in pipeline stage 3 is a valid instruction. The
-- instruction may not be valid if a junk instruction has
-- been allowed to come into the pipeline in order to allow
-- the pipeline to continue running when an instruction
-- cannot be fetched by the memory controller, or when an
-- instruction has been killed.
--
-- in p3int U This signal indicates that an interrupt jump instruction
-- (fantasy instruction) is currently in stage 3. This signal

```

```

--          causes (in conjunction with p3ilevl) the appropriate
--          interrupt mask bits to be cleared in the status register.
--          Note that p3int and p3iv are mutually exclusive.
--
-- in  p3ilevl      U This is used in conjunction with p3int to indicate which
--                  level of interrupt is being processed, and hence which of
--                  the interrupt mask bits should be cleared.
--                  It comes from bit 7 of the jump instruction word, which is
--                  set when a levell (lowest level) interrupt is being
--                  processed.
--
-- out p3condtrue   U This signal is produced from the result of the internal
--                  stage 3 condition code unit or from an extension cc unit
--                  (if implemented). A bit (bit 5) in the instruction selects
--                  between the internal and extension cc unit results. In
--                  addition, this signal is set true if the instruction is
--                  using short immediate data. As it is only used by
--                  flags.vhd in conjunction with the p3i=oflag, and
--                  with p3setflags, it does not include a decode for
--                  instructions which do not have a condition code field
--                  (i.e. all load and store operations).
--                  Does not include p3iv.
--
-- out p3setflags   U This signal is used by regular alu-type instructions and
--                  the jump instruction to control whether the supplied flags
--                  get stored. It is produced from the set-flags bit in the
--                  instruction word, but if that field is not present in the
--                  instruction (e.g. short immediate data is being used)
--                  then it will either come from the set-flag modes implied
--                  by which short immediate data register is used, or it will
--                  be set false if the instruction does not affect the flags.
--                  Does not include p3iv.
--
-- out p3cc[3:0]    L This bus contains the region of the instruction which
--                  contains the four-bit condition code field. It is sent
--                  with the alu flags to the extension condition code test
--                  logic which provides in return a signal (xp3ccmatch)
--                  which indicates whether it considers the condition to be
--                  true. The ARC decides whether to use the internal
--                  condition-true signal or the signal provided by extensions
--                  depending on the fifth bit of the instruction. This is
--                  handled within rctl.vhd.
--
-- in  xp3ccmatch   U This signal is provided by an extension condition-code
--                  unit which takes the condition code field from the
--                  instruction (at stage 3), and the alu flags (from stage 3)
--                  performs some operation on them and produces this
--                  condition true signal. Another bit in the instruction word
--                  indicates to the ARC whether it should use the internal
--                  condition-true signal or the one provided by the extension
--                  logic. This technique will allow extra ALU instruction
--                  conditions to be added which may be specific to different
--                  implementations of the ARC.
--
-- out sc_reg1      U This signal is produced by the pipeline control unit rctl,
--                  and is set true when an instruction in stage 3 is going to
--                  generate a write to the register being read by source 1 of

```

```

-- the instruction in stage 2. This is a source 1 shortcut.
-- It is used by the core register module to switch the stage 3
-- result bus onto the stage 2 source 1 result.
-- Extension core registers can have shortcutting banned
-- if x_p2noscl is set true at the appropriate time.
-- Includes both p2iv and p3iv.
-- The lasts1 signal is sc_reg1 and sc_load1 ORed together.
--
-- out sc_load1    U This signal is set true when data from a returning load is
--                  required to be shortcut onto the stage 2 source 1 result
--                  bus. This will only be the case if fast-load-returns are
--                  enabled, or if a four-port register file is used. If the 4p
--                  register file is implemented, the data used for the shortcut
--                  comes direct from the memory system, this requiring an
--                  additional input into the shortcut muxer.
--                  Extension core registers can have shortcutting banned
--                  if x_p2noscl is set true at the appropriate time.
--                  Includes both p2iv and p3iv.
--                  The lasts1 signal is sc_reg1 and sc_load1 ORed together.
--
-- out sc_reg2     U This signal is produced by the pipeline control unit rctl,
--                  and is set true when an instruction in stage 3 is going to
--                  generate a write to the register being read by source 2 of
--                  the instruction in stage 2. This is a source 1 shortcut.
--                  It is used by the core register module to switch the stage 3
--                  result bus onto the stage 2 source 2 result.
--                  Extension core registers can have shortcutting banned
--                  if x_p2nosc2 is set true at the appropriate time.
--                  Includes both p2iv and p3iv.
--                  The lasts2 signal is sc_reg2 and sc_load2 ORed together.
--
-- out sc_load2    U This signal is set true when data from a returning load is
--                  required to be shortcut onto the stage 2 source 2 result
--                  bus. This will only be the case if fast-load-returns are
--                  enabled, or if a four-port register file is used. If the 4p
--                  register file is implemented, the data used for the shortcut
--                  comes direct from the memory system, this requiring an
--                  additional input into the shortcut muxer.
--                  Extension core registers can have shortcutting banned
--                  if x_p2nosc2 is set true at the appropriate time.
--                  Includes both p2iv and p3iv.
--                  The lasts2 signal is sc_reg2 and sc_load2 ORed together.
--
-- out p3dolink    L This signal is latched (with en2) from p2dolink which is
--                  true when a JLcc or branch-and-link instruction was taken,
--                  indicating that the link register needs to be stored. It
--                  is used by alu.vhd to switch the program counter value
--                  which has been passed down the pipeline onto the p3result
--                  bus. If this signal is to be used to give a fully qualified
--                  indication that a J/BLcc is in stage 3, it must be qualified
--                  with p3iv to take account of pipeline tearing between
--                  stages 2 and 3 which could cause the instruction in stage
--                  three to be repeated.
--
-- out p3dolink    L This signal is latched (with en2) from p2dolink which is
--                  true when a JLcc or branch-and-link instruction was taken,
--                  indicating that the link register needs to be stored. It

```



```

-- is used by alu.vhd to switch the program counter value
-- which has been passed down the pipeline onto the p3result
-- bus. If this signal is to be used to give a fully qualified
-- indication that a J/BLcc is in stage 3, it must be qualified
-- with p3iv to take account of pipeline tearing between
-- stages 2 and 3 which could cause the instruction in stage
-- three to be repeated.
--
-- out p3lr      U This signal is used by hostif.vhd. It is produced from
--                a decode of p3i[4:0], p3iw(13) (check for LR) and
--                includes p3iv. Also used in extension logic for separate
--                decoding of auxiliary accesses from host and ARC.
--
-- out p3sr      U This signal is used by hostif.vhd. It is produced from
--                a decode of p3i[4:0], p3iw(25) (check for SR) and
--                includes p3iv. Also used in extension logic for separate
--                decoding of auxiliary accesses from host and ARC.
--
-- out mload     U This signal indicates to the LSU that there is a valid
--                load instruction in stage 3. It is produced from a decode
--                of p3i[4:0], p3iw(13) (to exclude LR) and the p3iv signal.
--
-- out mstore    U This signal indicates to the LSU that there is a valid
--                store instruction in stage 3. It is produced from a decode
--                of p3i[4:0], p3iw(25) (to exclude SR) and the p3iv signal.
--
-- out size[1:0] L This pair of signals are used to indicate to the LSU
--                the size of the memory transaction which is being
--                requested by a LD or ST instruction. It is produced
--                during stage 2 and latched as the size information bits
--                are encoded in different places on the LD and ST
--                instructions. It must be qualified by the mload/mstore
--                signals as it does not include an opcode decode.
--
-- out sex       L This signal is used to indicate to the LSU whether
--                a sign-extended load is required. It is produced during
--                stage 2 and latched as the sign-extend bit in the two
--                versions of the LD instruction (LDO/LDR) are in different
--                places in the instruction word.
--
-- out nocache   L This signal is used to indicate to the LSU whether
--                the load/store operation is required to bypass the cache.
--                It comes from bit 5 of the ld/st control group which is
--                found in different places in the ldo/ldr/st instructions.
--
-- out ldvalid_wb U This signal is used to control the switching of returning
--                load data onto the writeback path for the register file.
--                It is set true whenever returning load data must pass
--                through the regular load writeback path - this will be
--                loads to r32-r60 for a 4p regfile system, or loads to
--                r0-r60 for a 3p regfile system.
--
-- in ldvalid    U From LSU. This signal is set true by the LSU to
--                indicate that a delayed load writeback WILL occur on
--                the next cycle. If the instruction in stage 3 wishes to
--                perform a writeback, then pipeline stage 1, 2 and 3 will
--                be held. If the instruction in stage 3 is invalid, or

```

```

--      does not want to write a value into the core register
--      set for some reason, and fast-load-returns are enabled,
--      then the instructions in stages 1 and 2 will move into
--      2 and 3 respectively, and the instruction that was in
--      stage 3 will be replaced in stage 4 by the delayed load
--      writeback.
--      ** Note that delayed load writebacks WILL complete, even
--      if the processor is halted (en=0). In this instance, the
--      host may be held off for a cycle (hold_host) if it is
--      attempting to access the core registers. **
--
--      in  regadr[5:0] U From LSU. This bus carries the address of the register
--                     into which the delayed load will writeback when ldvalid
--                     is true. rctl.vhd will ensure that this value is latched
--                     onto wba[5:0] at the end of a cycle when ldvalid is true,
--                     even cycles when the processor is halted (en = 0).
--
--      in  mwait      U From MC. This signal is set true by the MC in order
--                     to hold up stages 1, 2, and 3. It is used when the
--                     memory controller cannot service a request for a memory
--                     access which is being made by the LSU. It will be
--                     produced from mload, mstore and logic internal to the
--                     memory controller.
--
--      in  xshimm      U From extensions. Indicates that an extension instruction
--                     in stage 3 is using short-immediate data other than that
--                     implied by the use of one of the short-immediate data
--                     registers. It is used by rctl to ensure correct values for
--                     p3condtrue and p3setflags are generated. Qualified by
--                     x_idcode3, xt_aluop and p3iv (eventually).
--
--      in  xholdup123  U From extensions. This is used by extension ALU
--                     instructions to hold up the pipeline if the function
--                     requested cannot be completed on the current cycle.
--                     Pipeline stages 1, 2 and 3 will typically be held, but the
--                     writeback (stage 4) will continue.
--
--      in  x_idcode3   L From extensions. This signal will be true when the
--                     extension logic detects an extension instruction in
--                     stage 3. It is latched from x_idcode2 by the extensions
--                     when en2 is true at the end of a cycle.
--                     It is used to correctly generate p3condtrue, p3setflags,
--                     and to detect (along with xnwb) when a register writeback
--                     will take place.
--
--      in  xnwb        U From extensions. Extension instructions utilise the
--                     normal writeback-control logic (ins.cc's, dest=imm,
--                     short imm data etc), but in addition have extra functions.
--                     When the extension logic has 'claimed' an instruction
--                     in stage 3 by setting x_idcode3, it can also disable
--                     writeback for that instruction by setting xnwb. When
--                     x_idcode3 is low, or if the instruction is 'claimed' by
--                     the ARC, xnwb has no effect.
--
--      in  (x_ialusel) U From extensions. This signal is provided to allow
--                     extension instructions to utilise basecase ALU operations
--                     for their own purposes. This is intended to be used to

```

```

-- (not req. by) load up fifo command buffers for pixel engines etc.
-- (rctl, here ) The ALU only decodes the bottom four bits of the
-- (for info ) instruction opcode directly, and has extra logic to take
-- (only. ) account of the x_ialusel signal.
-- If extensions want to shadow an ALU operation, the top
-- bit is set (ie an extension instruction), whilst the
-- rest of the instruction is set up as per the basecase ALU
-- instruction. The ALU result mux will select an internal
-- ALU result if i4 = 0 (basecase instruction), or if
-- i4=1 (extension), x_ialusel=1 (use int. result),
-- x_idecode3=1 (valid extension instruction). The extension
-- logic should also set xnwb to prevent writeback to the
-- core register set. Flag setting will work normally unless
-- the xsetflags signal is set, in which case the flags
-- will be loaded from the xflags[3:0] bus.
-- xp2idest should be set when the instruction is in stage 2
-- to prevent the scoreboard unit from checking the dest
-- register field.

```

```

----- Breakpoint/Actionpoint signals -----

```

```

-- in actionhalt This signal is set true when the
-- actionpoint (if selected) has been triggered by a valid
-- condition. The ARC pipeline is halted and flushed when
-- this signal is '1'.

```

```

-- Note: The pipeline is flushed of instructions when the
-- breakpoint instruction is detected, and it is important
-- to disable each stage explicitly. A normal instruction in
-- stage one will mean that instructions in stage two, three
-- and four will be allowed to complete. However, for an
-- instruction in stage one which is in the delay slot of a
-- branch, loop or jump instruction means that stage two
-- has to be stalled as well. Therefore, only stages three
-- and four will be allowed to complete.

```

```

-- out brk_inst U To flags.vhd. This signals to the ARC that a breakpoint
-- instruction has been detected in stage one of the
-- pipeline. Hence, the halt bit in the flag register has to
-- be updated in addition to the BH bit in the debug
-- register. The pipeline is stalled when this signal is set
-- to '1'.

```

```

-- Note: The pipeline is flushed of instructions when the
-- breakpoint instruction is detected, and it is important
-- to disable each stage explicitly. A normal instruction in
-- stage one will mean that instructions in stage two, three
-- and four will be allowed to complete. However, for an
-- instruction in stage one which is in the delay slot of a
-- branch, loop or jump instruction means that stage two
-- has to be stalled as well. Therefore, only stages three
-- and four will be allowed to complete.

```

```

-- out AP_p3disable_r This signals to the ARC that the
-- pipeline has been flushed due to a breakpoint or sleep
-- instruction. If it was due to a breakpoint instruction
-- the ARC is halted via the 'en' bit, and the AH bit is

```

```

--          set to '1' in the debug register.
--
-- out p2limm          This is used by the actionpoint
--                      debugging system when selected to qualify the value of
--                      the PC at stage one of the pipeline. The limm data is
--                      considered to be at the same the value address as the
--                      instruction it is associated with regards to the
--                      debugger.
--
----- Sleep Mode signals -----
--
-- in  sleeping        This is the sleep mode flag ZZ in the debug register
--                      (bit 23). When it is true the ARC is stalled. This flag
--                      is set when the p2sleep_inst is true and
--                      cleared on restart or interrupt.
--
-- out p2sleep_inst    This signal is set when a sleep instruction has been
--                      decoded in pipeline stage 2. It is used to set the sleep
--                      mode flag ZZ (bit 23) in the debug register.
--
----- Instruction Step signals -----
--
-- in  do_inst_step    This signal is set when the single step flag (SS) and the
--                      instruction step flag (IS) in the debug register has been
--                      written to simultaneously through the host interface. It
--                      indicates that an instruction step is being performed.
--                      When the instruction step has finished this signal goes
--                      low.
--
-- out stop_step       This signal is set when the instruction step has
--                      finished.
--
-----
ENTITY rctl IS
PORT(  signal  ck          : in  std_ulogic;          -- system clock
       signal  clr        : in  std_ulogic;          -- system reset
       signal  en         : in  std_ulogic;          -- system go

-----** Stage 1 **-----

       signal  pliw        : in  std_ulogic_vector(31 downto 0);
       signal  ivalid      : in  std_ulogic;
       signal  ivic        : in  std_ulogic;
       signal  pcen        : out std_ulogic;
       signal  ifetch      : out std_ulogic;
       signal  ipending    : out std_ulogic;
       signal  en1         : out std_ulogic;
       signal  plint       : in  std_ulogic;

-----** Stage 2 **-----

       signal  en2         : out std_ulogic;
       signal  p2i         : out std_ulogic_vector(4 downto 0);
       signal  p2iv        : out std_ulogic;
       signal  fs1a        : out std_ulogic_vector(5 downto 0);

```

```

signal s2a          : out std_ulogic_vector(5 downto 0);
signal dest         : out std_ulogic_vector(5 downto 0);
signal slen         : out std_ulogic;
signal s2en         : out std_ulogic;
signal xholdup12    : in  std_ulogic;
signal desten       : out std_ulogic;
signal xp2idest     : in  std_ulogic;
signal x_idecode2   : in  std_ulogic;
signal p2shimm      : out std_ulogic_vector(8 downto 0);
signal p2offset     : out std_ulogic_vector(19 downto 0);
signal p2cc         : out std_ulogic_vector(3 downto 0);
signal xp2ccmatch   : in  std_ulogic;
signal p2condtrue   : out std_ulogic;
signal p2setflags   : out std_ulogic;
signal p2int        : in  std_ulogic;
signal p2jblcc      : out std_ulogic;
signal p2st         : out std_ulogic;
signal p2ldo        : out std_ulogic;
signal p2lr         : out std_ulogic;
signal mload2       : out std_ulogic;
signal mstore2      : out std_ulogic;
signal holdup12     : in  std_ulogic;
signal aluflags     : in  std_ulogic_vector(3 downto 0);
signal x_p2noscl    : in  std_ulogic;
signal x_p2nosc2    : in  std_ulogic;
signal dorel        : out std_ulogic;
signal dojcc        : out std_ulogic;
signal p2killnext   : out std_ulogic;

```

-----\*\* Stage 3 \*\*-----

```

signal en3          : out std_ulogic;
signal p3i          : out std_ulogic_vector(4 downto 0);
signal p3a          : out std_ulogic_vector(5 downto 0);
signal p3c          : out std_ulogic_vector(5 downto 0);
signal p3iv         : out std_ulogic;
signal p3int        : in  std_ulogic;
signal p3ilev1      : in  std_ulogic;
signal p3condtrue   : out std_ulogic;
signal p3setflags   : out std_ulogic;
signal p3cc         : out std_ulogic_vector(3 downto 0);
signal xp3ccmatch   : in  std_ulogic;
-- signal lasts1    : out std_ulogic;
-- signal lasts2    : out std_ulogic;
signal sc_reg1      : out std_ulogic;
signal sc_reg2      : out std_ulogic;
signal sc_load1     : out std_ulogic;
signal sc_load2     : out std_ulogic;
signal p3dolink     : out std_ulogic;
signal p3lr         : out std_ulogic;
signal p3sr         : out std_ulogic;
signal mload        : out std_ulogic;
signal mstore       : out std_ulogic;
signal size         : out std_ulogic_vector(1 downto 0);
signal sex          : out std_ulogic;
signal nocache      : out std_ulogic;
signal ldvalid      : in  std_ulogic;

```

```

signal  regadr      : in  std_ulogic_vector(5 downto 0);
signal  mwait       : in  std_ulogic;
signal  xshimm      : in  std_ulogic;
signal  xholdup123  : in  std_ulogic;
signal  x_idcode3    : in  std_ulogic;
signal  xnwb        : in  std_ulogic;
signal  p3wb_en     : out std_ulogic;
signal  p3wb_nxt    : out std_ulogic;
signal  p3wba       : out std_ulogic_vector(5 downto 0);

signal  p3_ni_wbrq   : out std_ulogic;
signal  ldvalid_wb   : out std_ulogic;

```

-----\*\* Debug interface \*\*-----

```

signal  actionhalt   : in  std_ulogic;
signal  hw_brk_only  : in  std_ulogic;
signal  sleeping     : in  std_ulogic;
signal  do_inst_step : in  std_ulogic;
signal  stop_step    : out std_ulogic;
signal  p2sleep_inst : out std_ulogic;
signal  brk_inst     : out std_ulogic;
signal  p2limm       : out std_ulogic;
signal  AP_p3disable_r : out std_ulogic;

signal  p2limm_data_r : out std_ulogic_vector(31 downto 0);
signal  fetch_rolling_r : in  std_ulogic;
signal  p2merge_valid_r : out std_ulogic;

```

END rctl;

-----

ARCHITECTURE synthesis OF rctl IS

-- internal signals:

```

SIGNAL i_ifetch      : std_ulogic;
SIGNAL ipcen         : std_ulogic;
SIGNAL ien1          : std_ulogic;
SIGNAL ien1_lowpower : std_ulogic;

```

-- for debugging and halting the pipeline stages

```

SIGNAL i_brk_decode   : std_ulogic;
SIGNAL i_brk_inst     : std_ulogic;
SIGNAL i_brk_pass     : std_ulogic;
SIGNAL i_kill_AP      : std_ulogic;
SIGNAL i_break_stage1 : std_ulogic;
SIGNAL i_break_stage2 : std_ulogic;
SIGNAL i_AP_p2disable_r : std_ulogic;
SIGNAL i_AP_p3disable_r : std_ulogic;
SIGNAL i_n_AP_p2disable : std_ulogic;

```

```

SIGNAL i_n_AP_p3disable : std_ulogic;
SIGNAL ip2sleep_inst    : std_ulogic;
signal istop_step       : std_ulogic;
signal inst_stepping    : std_ulogic;
signal plp2step         : std_ulogic;
signal p2step           : std_ulogic;
signal p3step           : std_ulogic;
signal pcen_step        : std_ulogic;

SIGNAL ien2             : std_ulogic;
SIGNAL ip2iw            : std_ulogic_vector(31 downto 0);
SIGNAL ip2i             : std_ulogic_vector(4 downto 0);
SIGNAL ip2a             : std_ulogic_vector(5 downto 0);
SIGNAL ip2b             : std_ulogic_vector(5 downto 0);
SIGNAL ip2c             : std_ulogic_vector(5 downto 0);
SIGNAL ip2q             : std_ulogic_vector(4 downto 0);
SIGNAL ip2dd            : std_ulogic_vector(1 downto 0);
SIGNAL ip2ld            : std_ulogic;
SIGNAL ip2_fbit         : std_ulogic;
SIGNAL ip2iv            : std_ulogic;
SIGNAL ip2ccmatch       : std_ulogic;
SIGNAL ip2condtrue      : std_ulogic;
SIGNAL ishi_bf          : std_ulogic;
SIGNAL ishi_bn          : std_ulogic;
SIGNAL ishi_cf          : std_ulogic;
SIGNAL ishi_cn          : std_ulogic;
SIGNAL ip2shimm         : std_ulogic;
SIGNAL ip2shimmf        : std_ulogic;
SIGNAL islen            : std_ulogic;
SIGNAL is2en            : std_ulogic;
SIGNAL idesten          : std_ulogic;
SIGNAL ip2jblcc         : std_ulogic;
SIGNAL ip2mop_e         : std_ulogic_vector(memop_esz downto 0);
SIGNAL ip2size          : std_ulogic_vector(1 downto 0);
SIGNAL ip2sex           : std_ulogic;
SIGNAL ip2awb           : std_ulogic;
SIGNAL ip2nocache       : std_ulogic;
SIGNAL ip2ldo           : std_ulogic;
SIGNAL ip2st            : std_ulogic;
SIGNAL ip2limm          : std_ulogic;
SIGNAL ip2bch           : std_ulogic;
SIGNAL ip2killnext      : std_ulogic;
SIGNAL ip2pldep         : std_ulogic;
SIGNAL ip2rjmp          : std_ulogic;
SIGNAL ip2jumping       : std_ulogic;
SIGNAL ip2nojump        : std_ulogic;
SIGNAL ip2lpcc          : std_ulogic;
SIGNAL ip2dolink        : std_ulogic;
SIGNAL ilasts1          : std_ulogic;
SIGNAL ilasts2          : std_ulogic;

SIGNAL ien3             : std_ulogic;
SIGNAL ip3i             : std_ulogic_vector(4 downto 0);
SIGNAL ip3a             : std_ulogic_vector(5 downto 0);
SIGNAL ip3b             : std_ulogic_vector(5 downto 0);
SIGNAL ip3c             : std_ulogic_vector(5 downto 0);
SIGNAL ip3q             : std_ulogic_vector(4 downto 0);

```

```

SIGNAL ip3_fbit          : std_ulogic;
SIGNAL ip3shimm          : std_ulogic;
SIGNAL ip3shimmf        : std_ulogic;
SIGNAL ip3iv            : std_ulogic;
SIGNAL ip3ccmatch       : std_ulogic;
SIGNAL ip3condtrue      : std_ulogic;
SIGNAL ip3setflags      : std_ulogic;
SIGNAL ip3size          : std_ulogic_vector(1 downto 0);
SIGNAL ip3sex           : std_ulogic;
SIGNAL ip3awb           : std_ulogic;
SIGNAL ip3nocache       : std_ulogic;
SIGNAL ip3wb_en         : std_ulogic;
SIGNAL ip3dolink        : std_ulogic;
SIGNAL ip3dimm          : std_ulogic;
SIGNAL imload3          : std_ulogic;
SIGNAL imstore3         : std_ulogic;
SIGNAL ip3m_awb         : std_ulogic;
SIGNAL ip3ccwb_op       : std_ulogic;
SIGNAL ip3xwb_op        : std_ulogic;
SIGNAL ip3_wb_req       : std_ulogic;
SIGNAL ip3_wb_rsv       : std_ulogic;
SIGNAL ip3lir           : std_ulogic;
SIGNAL ip3sr            : std_ulogic;

SIGNAL ip3wba           : std_ulogic_vector(5 downto 0);
SIGNAL ip3_sc_wba       : std_ulogic_vector(5 downto 0);
SIGNAL iwben            : std_ulogic;

SIGNAL new_p2iw         : std_ulogic_vector(31 downto 0);
SIGNAL new_p3i          : std_ulogic_vector(opcodsz downto 0);
SIGNAL new_p3a          : std_ulogic_vector(oprandsz downto 0);
SIGNAL new_p3b          : std_ulogic_vector(oprandsz downto 0);
SIGNAL new_p3c          : std_ulogic_vector(oprandsz downto 0);
SIGNAL new_p3q          : std_ulogic_vector(qqsiz downto 0);
SIGNAL new_p3_fbit      : std_ulogic;
SIGNAL new_p3shimm      : std_ulogic;
SIGNAL new_p3shimmf     : std_ulogic;
SIGNAL new_p3size       : std_ulogic_vector(1 downto 0);
SIGNAL new_p3sex        : std_ulogic;
SIGNAL new_p3awb        : std_ulogic;
SIGNAL new_p3nocache    : std_ulogic;
SIGNAL new_p3dolink     : std_ulogic;
SIGNAL new_wba          : std_ulogic_vector(oprandsz downto 0);
SIGNAL iwba             : std_ulogic_vector(oprandsz downto 0);

SIGNAL n_p2iv           : std_ulogic;
SIGNAL n_p3iv           : std_ulogic;
SIGNAL i_awake          : std_ulogic;
SIGNAL l_go             : std_ulogic;
SIGNAL n_go             : std_ulogic;
SIGNAL ni_go            : std_ulogic;
SIGNAL i_hostload       : std_ulogic;
SIGNAL ip3wb_nxt        : std_ulogic;

SIGNAL ip2limm1         : std_ulogic;
SIGNAL ip2limm2         : std_ulogic;

```



```

SIGNAL ien3_non_iv      : std_ulogic;
SIGNAL ien2_non_iv      : std_ulogic;

SIGNAL ip3_load_stall   : std_ulogic;
SIGNAL isc_reg1         : std_ulogic;
SIGNAL isc_reg2         : std_ulogic;
SIGNAL isc_load1        : std_ulogic;
SIGNAL isc_load2        : std_ulogic;

SIGNAL ihp2_ld_nsc1     : std_ulogic;
SIGNAL ihp2_ld_nsc2     : std_ulogic;
SIGNAL ihp2_ld_nsc      : std_ulogic;

SIGNAL ibch_holdp2      : std_ulogic;
SIGNAL ibch_p3flagset   : std_ulogic;

SIGNAL ildvalid_wb      : std_ulogic;
signal ip2ivalid_r : std_ulogic;
signal ip2limm_data_r : std_ulogic_vector(31 downto 0);
signal i_p2merge_valid_r : std_ulogic;
signal i_fst_ifetch_r : std_ulogic;
signal i_p2_fst_ifetch_r : std_ulogic;
signal i_fetchen : std_ulogic;
signal i_pending_kill_r : std_ulogic;
signal i_cancel_kill_r : std_ulogic;
signal i_ifetch_r : std_ulogic;
signal i_ipending : std_ulogic;
signal i_p1_used_r : std_ulogic;

```

```

BEGIN

```

```

    -- New Outputs

```

```

    --

```

```

    p2limm_data_r <= ip2limm_data_r;

```

```

    p2merge_valid_r <= i_p2merge_valid_r;

```

```

    ipending <= i_ipending;

```

```

    -----** Stage 1 **-----

```

```

merge_process : process(ck, clr)

```

```

    begin

```

```

        if clr = '1' then

```

```

            ip2ivalid_r <= '0';

```

```

            i_p2merge_valid_r <= '0';

```

```

            --PS

```

```

            ip2limm_data_r <= (others => '0');

```

```

            i_fst_ifetch_r <= '1';

```

```

            i_p2_fst_ifetch_r <= '1';

```

```

            i_pending_kill_r <= '0';

```

```

            i_cancel_kill_r <= '0';

```

```

            i_p1_used_r <= '0';

```

```

        elsif (ck'EVENT and ck = '1') then

```

```

-- Latch ivalid for use in stage 2
--
ip2ivalid_r <= ivalid;

-- Latch in long immediates when an instruction in stage 2
-- references a long immediate and its available in stage 1
-- Record that the long immediate is available and has been
-- merged with the opcode.
-- Indicate that the dataword in stage 1 has been used.
--
if ivalid = '1' and ip2limm = '1' then
    ip2limm_data_r <= pliw;
    i_p1_used_r <= '1';
    i_p2merge_valid_r <= '1';
end if;
-- Indicate that the dataword in stage 1 has been used.
--
if ien1_lowpower = '1' then
    i_p1_used_r <= '1';
end if;

-- When a new instruction dataword is requested clear the p1_
-- used flag
--
if i_ifetch = '1' then
    i_p1_used_r <= '0';
end if;

-- When the instruction in stage 2 moves and it references a lon
-- immediate clear the p2merge valid flag.
--
if ien2 = '1' and ip2limm = '1' then
    i_p2merge_valid_r <= '0';
end if;

-- Start up the pipeline so stage 1 can advance stage 0 when
-- stage 0 is stalled or an ivic is requested.
--
if ivic = '1' or i_ifetch = '0' then
    i_fst_ifetch_r <= '1';
    i_p2_fst_ifetch_r <= '1';
end if;

-- Clear the ifetch advancement flag
--
if i_ifetch = '1' then
    i_fst_ifetch_r <= '0';
end if;

-- Clear the ifetch advancement flag
if i_fst_ifetch_r = '0' and ivic = '0' then
    i_p2_fst_ifetch_r <= '0';
end if;

-- Re-intialize to ifetch advancement flags
-- since ifetching has been stalled

```

```

--
if ivic = '1' or i_ifetch = '0' then
    i_fst_ifetch_r <= '1';
    i_p2_fst_ifetch_r <= '1';
end if;

--
if ivalid = '0' and ip2killnext = '1' and ien2 = '0' then
    i_cancel_kill_r <= '1';
end if;

if ien2 = '1' then
    i_cancel_kill_r <= '0';
end if;

if ivalid = '0' and ip2killnext = '1' and ien2 = '1'
    and i_cancel_kill_r = '0' then
    i_pending_kill_r <= '1';
end if;

-- Clear the pending instruction kill flag when the instruction
-- is killed
if i_pending_kill_r = '1' and ivalid = '1' then
    i_pending_kill_r <= '0';
end if;

--Not used ...
i_ifetch_r <= i_ifetch;

end if;
end process;

```

```

--** Stage 1 logic **--

```

```

-- The breakpoint instruction is determined at stage 1 from:

```

```

-- [1] Decode of pliw,
-- [2] Instruction at stage 1 is valid,
-- [3] The instruction is not killed,
-- [4] The instruction is not long immediate data,
-- [5] There is no sleep instruction in stage 2.
--

```

```

i_brk_decode <= '1' WHEN (pliw(instrubnd downto instrlbnd) = oflag)
    AND (pliw(copubnd downto coplbnd) = so_brk)
    AND (pliw(shimmlbnd) = '0')
    AND (ip2ivalid_r = '1') else
    '0';

```

```

i_brk_pass    <= NOT(ip2killnext)    AND
    NOT(ip2limm)    AND
    NOT(ip2sleep_inst);

```

```

i_brk_inst    <= i_brk_decode AND i_brk_pass;

```

```

brk_inst      <= '0'; --i_brk_inst;

-----** Stage 2 **-----

--
-- The sleep instruction is determined at stage 2 from:
--
-- [1] Decode of p2iw,
-- [2] Instruction at stage 2 is valid.
--
--
ip2sleep_inst <= '1' WHEN (ip2iw(instrubnd downto instrlbnd) = oflag)
                        AND (ip2iw(copubnd downto coplbnd) = so_sleep)
                        AND (ip2iw(shimmlbnd) = '1')
                        AND (ip2iv = '1') ELSE
                        '0';

p2sleep_inst  <= ip2sleep_inst;

-----

-- The data to be used for input to stage 2 is latched here.
--
-- Clock the instruction presented by the memory controller when ien1 is
-- true. - Also clock p2iv, which is ivalid clocked when ien1 is true.
-- This signal is used to indicate which instructions in the pipeline are
-- real and which are junk which is being allowed to flow through to keep
-- things running.

-----
--- p2ins:      pipe32 PORT MAP (ck, ien1_lowpower, clr, pliw, ip2iw);

new_p2iw      <= pliw WHEN ien1_lowpower = '1' ELSE ip2iw;

p2ins : PROCESS(ck, clr)
BEGIN
  IF clr = '1' THEN
    ip2iw <= (others => '0');

  ELSIF (ck'EVENT AND ck = '1') THEN

    if ien1_lowpower = '1' and ip2limm = '0' then

      end if;

      ip2iw <= new_p2iw;
    END IF;
  END PROCESS;

-----

-- The various component parts of the instruction are extracted here to
-- internal signals.

ip2i          <= ip2iw(instrubnd downto instrlbnd);  -- opcode
ip2a          <= ip2iw(aopubnd downto aoplbnd);      -- a field
ip2b          <= ip2iw(bopubnd downto boplbnd);      -- b field

```

```

ip2c      <= ip2iw(copubnd downto coplbnd);      -- c field
ip2_fbit  <= ip2iw(setflgpos);                  -- flag bit
ip2q      <= ip2iw(qqubnd downto qqlbnd);        -- q field
ip2dd     <= ip2iw(ddubnd downto ddlbnd);        -- delay slot mode

-- Output drives of signals direct from the stage 2 input latch.
-- (some more extraction takes place also)

p2i       <= ip2i;                             -- opcode
dest      <= ip2a;                             -- destination
fsla      <= ip2b;                             -- source 1
s2a       <= ip2c;                             -- source 2
p2shimm   <= ip2iw(shimmubnd downto shimmlbnd);  -- short immediate
p2cc      <= ip2iw(ccubnd downto cclbnd);        -- CC field (no x bit)
p2offset  <= ip2iw(targubnd downto targlbnd);    -- branch offset

-- Now some simple decodes from the opcode field are performed.
-- These are for files which do their own decode of the p2i[] field.

ip2jblcc  <= '1' WHEN (ip2i = oblcc)             -- branch and link
           OR ((ip2i = ojcc) AND (ip2c(0) = '1')) -- jump and link.
           ELSE '0';

-- output drives --

p2jblcc   <= ip2jblcc;

ip2st     <= '1' WHEN (ip2i = ost) AND ip2iw(25) = '0' ELSE '0'; -- ST
instruction.

p2st      <= ip2st;

mstore2   <= ip2st AND ip2iv;

-- The load instruction has two opcodes ldr (00) and ldo (01). The aux LR
-- instruction is encoded on the ldo instruction, so must be excluded when
-- producing a signal which indicates that a load instruction is in stage 2.

ip2ld     <= '1' WHEN (ip2i = oldr)
           OR (ip2i = oldo AND ip2iw(13) = '0')   ELSE
           '0';

mload2    <= ip2ld AND ip2iv;

p2lr      <= '1' WHEN (ip2i = oldo) and ip2iw(13) = '1' ELSE '0';

ip2ldo    <= '1' WHEN (ip2i = oldo) and ip2iw(13) = '0' ELSE '0';

p2ldo     <= ip2ldo;

-- output drives --

slen      <= islen;
s2en      <= is2en;
desten    <= idesten;

```

```
-- Output drive --
```

```
p2condtrue <= ip2condtrue;
```

```
-- Stage 2 flag setting calculation --
```

```
--
-- p2setflags just comes from bit 8 of the instruction word. It is only
-- used in flags.vhd and is qualified there with a decode of p2i=ojcc,
-- and a check for p2iv and p2condtrue.
```

```
p2setflags <= ip2_fbit;
```

```
-- Produce signals to pass down the pipeline which indicate to stage 3
-- which register fields include immediate data registers, qualified with
-- the slen/s2en/desten signals.
```

```
--
-- Here four signals are produced, one for each of the combinations of
-- the two source fields and the two short immediate data registers
-- (i.e. set flags/don't set flags)
```

```
ishi_bf <= '1' WHEN ( ip2b = rfshimm ) and islen = '1' ELSE '0';
ishi_bn <= '1' WHEN ( ip2b = rnshimm ) and islen = '1' ELSE '0';
ishi_cf <= '1' WHEN ( ip2c = rfshimm ) and is2en = '1' ELSE '0';
ishi_cn <= '1' WHEN ( ip2c = rnshimm ) and is2en = '1' ELSE '0';
```

```
-- Now produce signals which indicate whether a short-imm field is present
-- at the bottom of the instruction, due to a register (ip2shimm),
-- and indicate whether the flags should be set or not (ip2shimmf).
```

```
ip2shimm <= ishi_bf OR ishi_bn OR ishi_cf OR ishi_cn;
ip2shimmf <= ishi_bf OR ishi_cf;
```

```
-- Now extract the extra encoding information used for loads and stores.
-- The signals are extracted and latched at the end of stage 2.
```

```
ip2mop_e <= ip2iw(ldo_eubnd downto ldo_elbnd) WHEN ip2ldo = '1' ELSE
ip2iw(st_eubnd downto st_elbnd) WHEN ip2st = '1' ELSE
ip2iw(ldr_eubnd downto ldr_elbnd);
```

```
ip2nocache <= ip2mop_e(ls_nc);
ip2size <= ip2mop_e(ls_subnd downto ls_slbnd);
ip2sex <= ip2mop_e(ls_ext);
ip2awb <= ip2mop_e(ls_awbck);
```

```
-- Generate signals for pipeline control and interrupt control units --
```

```
--
-- p2limm - this will be true when a valid instruction which uses long imm
-- data is in stage 2. Note that this signal will include p2iv as it includes
```

```
-- slen/s2en.
--
-- p2bch - this will be true when a jump instruction bcc/blcc/lpcc/jcc is
-- in stage 2. This also includes p2iv, but explicitly this time.
--
-- p2pldep - this signal is used to indicate that the instruction at stage 2
-- requires that the next instruction be in stage 1 before it can move off.
-- This may be either to ensure correct delay slot operation for a branch
-- or to make sure that long immediate data is fetched, and then killed
-- before it can be processed as an instruction.
--
```

```
ip2limm1 <= '1' WHEN ip2b = rlimm ELSE '0';
ip2limm2 <= '1' WHEN ip2c = rlimm ELSE '0';
```

```
ip2limm      <= (ip2limm1 AND islen) OR (ip2limm2 AND is2en);
```

```
ip2bch      <= '1' WHEN ip2iv = '1' AND (ip2i = obcc OR   ip2i = oblcc
                                         OR ip2i = olpcc OR   ip2i = ojcc )
```

```
ELSE
    '0';
```

```
ip2pldep    <= ip2limm OR ip2bch;
p2limm      <= ip2limm;
p2pldep     <= ip2pldep;
```

```
-----** Pipeline control unit **-----
```

```
-- ivalid      U From memory controller. Indicates that the instruction/data
--              word presented to the ARC on pliw[31:0] is valid.
--
-- plint       U Indicates that an interrupt has been detected, and an
--              interrupt-op will be inserted into stage 2 on the next
--              cycle, setting p2int true. This signal will have the
--              effect of canceling the instruction currently being
--              fetched by stage 1 by causing p2iv to be set false at the
--              end of the cycle when plint is true.
--
-- p2int       L Indicates that an interrupt-op instruction is in
--              stage 2. This signal is used in coreregs.vhd to control
--              the placing of the pc onto a source bus for writing back
--              to the interrupt link registers, and by aux_regs to
--              insert the interrupt vector int_vec[] into the program
--              counter, thus requiring this file to set pcen true.
--
-- p2bch       U This signal indicates that the instruction in stage 2 is
--              a branch or jump instruction, and therefore requires that
--              the instruction following must be present in the delay slot
--              before it can move on.
--              (Simple decode of p2i[4:0], and does include p2iv)
--
-- p2limm      U This signal indicates that the instruction in stage 2 uses
--              long immediate data for one of the source operands. This means
--              that the instruction cannot complete until the correct data
```

```

-- word has been fetched into stage 1. When the instruction does
-- move out of stage 2, the data word is marked as an invalid
-- instruction before it gets into stage 2. The data word has
-- served its purpose by this point, so it can be overwritten
-- by another instruction if stage 3 is stalled, and stage 1 is
-- allowed to move on into stage 2 over the top of the data word.
-- This signal includes slen/s2en and p2iv.
--
-- holdup12 U From lsu.vhd. This signal is used to hold up pipeline
-- stages 1 and 2 (pcen, en1 and en2) when the load store unit
-- finds a register being used by the instruction at stage
-- 2 which is the destination of a delayed load. It will
-- also be set when the scoreboard unit is full and the
-- ARC attempts to do another load. Stages 3 and 4 will
-- will continue running.
--
-- xholdup12 U From extensions. This signal is used to hold up pipeline
-- stages 1 and 2 (pcen, en1 and en2) when extension logic
-- requires that stage 2 be held up. For example, a core register
-- is being used as a window into SRAM, and the SRAM is not
-- available on this cycle, as a write is taking place from
-- stage 4, the writeback stage. Hence stage 2 must be held to
-- allow the write to complete before the load can happen.
-- Stages 3 and 4 will continue running.
--
-- p2killnext U This signal indicates that the delay slot mechanism of the
-- jump instruction currently in stage 2 is requesting that the
-- next instruction be killed before it gets into stage 2.
-- This signal is produced from a decode for a jump instruction
-- code, the condition-true signal, p2iv and the delay-slot field
-- in the instruction. This signal relies on the delay slot
-- instruction being present in stage 1 before stage 2 can move
-- on. This is handled elsewhere by this file.
--
-- ldvalid U From LSU. This signal is set true by the LSU to
-- indicate that a delayed load writeback WILL occur on
-- the next cycle. If the instruction in stage 3 wishes to
-- perform a writeback, then pipeline stage 1, 2 and 3 will
-- be held. If the instruction in stage 3 is invalid, or
-- does not want to write a value into the core register
-- set for some reason, then the instructions in stages 1
-- and 2 will move into 2 and 3 respectively, and the
-- instruction that was in stage 3 will be replaced in
-- stage 4 by the delayed load writeback.
-- ** Note that delayed load writebacks WILL complete, even
-- if the processor is halted (en=0). In this instance, the
-- host may be held off for a cycle (hold_host) if it is
-- attempting to access the core registers. **
--
-- mwait U From MC. This signal is set true by the MC in order
-- to hold up stages 1, 2, and 3. It is used when the
-- memory controller cannot service a request for a memory
-- access which is being made by the LSU. It will be
-- produced from mload3, mstore3 and logic internal to the
-- memory controller.
--
-- mload3 U This signal indicates to the LSU that there is a valid

```



```

-- load instruction in stage 3. It is produced from a decode
-- of p3i[4:0], p3iw(13) (to exclude LR) and the p3iv signal.
-- It is used here to ensure that a lockup situation cannot
-- occur when a branch is holding up stages 1, 2 and 3.
--
-- xholdup123 U From extensions. This is used by extension ALU
-- instructions to hold up the pipeline if the function
-- requested cannot be completed on the current cycle.
-- Pipeline stages 1, 2 and 3 will be held, but the
-- writeback (stage 4) will continue.
--
-- p3_wb_req U This signal (produced by rctl.vhd) is set true when the
-- instruction in stage 3 wants to writeback to the register
-- file, i.e. -
--     a. A destination register is given (r0-r60)
--     b. A link register to be written (interrupt, BLcc)
--     c. LD/ST with .A specified - to do address writeback
--
-- It will be false when no destination is required, i.e.
--     a. jumps/branches (not BLcc)
--     b. instructions with dest = immediate
--     c. instructions for which the condition is false
--     d. LD/ST without .A specified - no address writeback
--     e. cancelled instructions (p3iv = '0')
--     f. extension instruction, xnwb = '1'
--
-- p3_wb_rsv U This signal is set true when the instruction at stage 3
-- wants to reserve the writeback stage for itself. This is
-- required when a FIFO-type instruction wants to suppress
-- writeback to the register file, but needs the data and
-- register address to be present in the writeback stage so that
-- it can be picked off and sent into the FIFO buffer.
-- Is it generated by rctl.vhd and will be true when an
-- extension instruction at stage 3 is suppressing writeback
-- with the xnwb signal.
--
-- cr_hostw U This signal is set true to indicate that a host write
-- to the core registers will take place on the next cycle,
-- and that the end-of-stage 3 data and register address latches
-- should clock in the address and data provided by the host.
-- *** Note that host writes are overridden by returning delayed
-- loads. This signal hold_host will be asserted (produced in
-- rctl.vhd) to tell the host to wait for a cycle. ***
--
-- h_pcwr U From pcounter.vhd. This signal is set true when the host
-- is attempting to write to the pc/status register, and the
-- ARC is stopped. It is used to trigger an instruction fetch
-- when the PC is written when the ARC is stopped. This is
-- necessary to ensure the correct instruction is executed
-- when the ARC is restarted.
--
-- Outputs:
--
-- en1 U Stage 2 pipeline latch control. True when an instruction
-- is being latched into pipeline stage 2. Will be true
-- at different times to pcen, as it allows junk instructions
-- to be latched into the pipeline.

```

```

--      *** A feature of this signal is that it will allow an
--      instruco be clocked into stage 2 even when stage 3
--      is halted, provided that stage 2 contains a killed instruction
--      (i.e. p2iv = '0'). This is called a 'catch-up'. ***
--
--      ifetch      U This signal, similar to pcen, indicates to the memory
--                  controller that a new instruction is required, and should
--                  be fetched from memory from the address which will be clocked
--                  into currentpc[25:2] at the end of the cycle. It is also
--                  true for one cycle when the processor has been started
--                  following a reset, in order to get the ball rolling.
--                  An instruction fetch will also be issued if the host changes
--                  the program counter when the ARC is halted, provided it is
--                  not directly after a reset.
--                  The ifetch signal will never be set true whilst the memory
--                  controller is in the process of doing an instruction fetch,
--                  so it may be used by the memory controller as an
--                  acknowledgement of instruction receipt.
--
--      ipending    U This signal is true when an instruction fetch has been
--                  issued, and it has not yet completed. It is not true directly
--                  after a reset before the ARC has started, as no instruction
--                  fetch will have been issued. It is used to hold off host
--                  writes to the program counter when the ARC is halted, as
--                  these accesses will trigger an instruction fetch.
--
--      pcen        U This signal is true when the pc is allowed to change state.
--                  It takes account of ivalid (stage 1 has fetched a valid
--                  instruction) and the interrupts which need to be able to
--                  prevent the pc from updating.
--                  *** A feature of this signal is that it will allow an
--                  instruction to be clocked into stage 2 even when stage 3
--                  is halted, provided that stage 2 contains a killed instruction
--                  (i.e. p2iv = '0'). This is called a 'catch-up'. ***
--
--      en2         U Stage 3 pipeline latch control. Controls transition of
--                  instruction in stage 2 to stage 3. Will be set false if the
--                  op in stage 2 requires data from stage 1 which is not
--                  forthcoming because the instruction cannot be fetched to
--                  stage 1 during this cycle (i.e. ivalid = '0'). This condition
--                  will occur for instructions which use long immediate data
--                  or for jump/branch instructions which require the correct
--                  instruction to be in the delay slot.
--
--      en3         U Stage 3 instruction completion control. This signal is set
--                  true to indicate that the instruction in stage 3 can complete
--                  at the end of the cycle and pass out of pipeline stage 3. It
--                  may or may not pass into stage 4 (the writeback stage),
--                  depending on whether a writeback is required or not. Taken
--                  on its own, this signal controls writeback to the flags.
--
--      p3wb_en     U Stage 4 pipeline latch control. Controls transition of the
--                  data on the p3result[31:0] bus, and the corresponding register
--                  address from stage 3 to stage 4. As these buses carry data
--                  not only from instructions but from delayed load writebacks
--                  and host writes, they must be controlled separately from the
--                  instruction in stage 3. This is because if the instruction in

```

```

-- stage 3 does not need to write a value back into a register,
-- and a delayed load writeback is about to happen, the
-- instruction is allowed to complete (i.e. set flags) whilst
-- the data from the load is clocked into stage 4. If however
-- the instruction in stage 3 DOES need to writeback to the
-- register file when a delayed load writeback is about to
-- happen, then the instruction in stage 3 must be held up
-- and not allowed to change the processor state, whilst the
-- data from the delayed load is clocked into stage 4 from
-- stage 3.
-- *** Note that p3wb_en can be true even when the processor
-- is halted, as delayed load writebacks and host writes use
-- this signal in order to access the core registers. ***
--
-- wben      L This signal is the stage 4 write enable signal. It is
--            latched from p3wb_en. Stage 4 is never held up.
--
-- p2iv      L Pipeline stage 2 instruction valid. This latched signal
--            indicates that the instruction in stage 2 is valid. When it
--            is set false, the instruction in stage 2 is either a junk
--            value clocked in to keep the pipeline running, or an
--            instruction which was killed by the interrupt system.
--
-- p3iv      L Pipeline stage 3 instruction valid. This latched signal
--            indicates that the instruction in stage 3 is valid. When it
--            is set false, the instruction in stage 3 is either a junk
--            value clocked in to keep the pipeline running, or an
--            instruction which was killed by the interrupt system, or
--            a blank slot inserted when the instruction in stage 2 was
--            not allowed to complete on the previous cycle. This blank
--            slot must be inserted otherwise the instruction which was
--            executed by stage 3 during the previous cycle will be executed
--            again during the current cycle.
--
----- pcen : Program counter update enable -----
--
-- This signal indicates to the program counter that a new value can be
-- loaded. This will be the case when:
--
-- a.   A valid instruction has been fetched and can be passed on to
--       stage 2, allowing the memory controller to start looking for the
--       next instruction to be executed.
--
--       *** Note that this logic handles the case when stage 2 contains
--       an invalid instruction which is held due to stall in stage 3, and
--       we allow the instruction in stage 1 to move into stage 2. ***
--
-- b.   An interrupt is in stage 2, and the interrupt vector is to be
--       clocked into the program counter. The instruction now being
--       fetched into stage 1 will be killed anyway, but we must wait
--       until it has been fetched to be sure that we do not issue a new
--       fetch request to the memory controller before the last one has
--       completed.
--       The interrupt vector should only be clocked into the program
--       counter when the interrupt can move out of stage 2. This will
--       ensure that the correct pc value will be placed in the interrupt

```

```

--          link register.
--
-- We will also want to forcibly prevent the program counter from being
-- updated in some cases:
--
--     a. An interrupt has been recognized, and we want to kill the
--         instruction currently in stage 1, and not increment the program
--         counter in order to ensure the correct PC value is stored into
--         the appropriate interrupt link register.
--
--     b. The breakpoint instruction (or valid actionpoint) is detected and
--         the pipeline is to be flushed, and then halted.
--
--     c. A single instruction step is being executed, whilst preventing
--         another ifetch from being generated in order to only execute one
--         instruction at a time. During a single instruction step the PC is
--         only allowed to be updated and (thereby generating a new ifetch)
--         when:
--
--         1.a valid instruction in stage 1 is allowed to pass into
--            stage 2.
--
--         2.a branch or jump instruction is in stage 2 has a killed
--            delay slot.
--
--         3.an instruction is in stage 2 that uses a long immediate.
--
--         4.an interrupt has been detected and is now in stage 2.
--
-- *** Note that if an invalid instruction in stage 2 is held (this will be
--      due to a stall at stage 3) then the instruction in stage 1 will be
--      allowed to move into stage 2. ***
--
--Added to allow pc updates to advance ifetching
-- ip2ivalid_r prevents the core advancing more than 1 cycle
-- i_p2_fst_ifetch_r = '1' and i_fst_ifetch_r = '0' allow the core to
-- initially advance ifetch to get thing 'rolling'
--
--
--          ipcen    <= '0'  WHEN en = '0'
--                      OR (ip2ivalid_r = '0' and not (i_p2_fst_ifetch_r = '1' and
-- i_fst_ifetch_r = '0'))
--                      --or (ip2limm = '1' and i_p2merge_valid_r = '1')
--                      OR (p2int = '1' AND ien2_non_iv = '0')
--                      OR (ip2iv = '1' AND ien2_non_iv = '0')
--                      OR (i_break_stagel = '1')
--                      or (ip2limm = '1' and ip2killnext = '1' and
-- i_p2merge_valid_r = '0' )
--                      OR inst_stepping = '1'
--                      OR plint = '1'
--                      ELSE
--                      '1';
--
------- inst_stepping : PC Disable for single instruction step -----
--
-- The signal inst_stepping prevents the PC from being updated, by disabling
-- the PC enable signal (pcen). The signal is set when a single instruction

```

```
-- step is being performed and the PC does not need to be updated
-- (pcen_step = '0').
--
```

```
inst_stepping <= '1' WHEN do_inst_step = '1'
                    AND pcen_step = '0'           ELSE
                    '0';
```

```
-- The signal pcen_step is set when a single instruction step is being
-- executed if the PC needs to be updated. This happens in the following
-- cases:
```

- a. A valid instruction in stage one is allowed to pass into stage 2.
- b. A branch or jump in stage 2 has a killed delay slot.
- c. An instruction using long immediate is in stage 2.
- d. An interrupt has been detected and is now in stage 2.

```
pcen_step <= '1' WHEN (do_inst_step = '1'
                      AND p2step = '0')
                      OR (p2step = '1'
                      AND (ip2killnext = '1'
                      OR ip2limm = '1'))
                      OR p2int = '1'
                      ELSE
                      '0';
```

```
----- stop_step : stop single instruction step when finished -----
```

```
-- The stop_step signal is related to single instruction step. When the
-- single instruction has been completed the stop_step signal goes high.
-- Depending on the type of instruction the stop is made in different
-- places in the pipeline:
```

- a. Branches and jumps with delay slots that are not killed stop in stage 2, because the instruction in the delay slot count as a new instruction. Next instruction step will execute the branch and the delay slot.
- b. All other instructions complete in stage 3 (if writeback is not performed) or stage 4 (if writeback is performed).

```
-- When the stop_step signal goes high the ARC is halted,
-- the step tracker signals (below) are reset and a new instruction fetch
-- is generated.
```

```
istop_step <= '1' WHEN (ip2bch = '1'
                      AND ip2limm = '0'
                      AND ip2killnext = '0'
                      AND p2step = '1')
                      OR (p3step = '1'
                      AND ip3wb_en = '0')
```

ELSE

'0';

stop\_step <= istop\_step;

----- step tracker : keeps track on the single step instruction -----

--

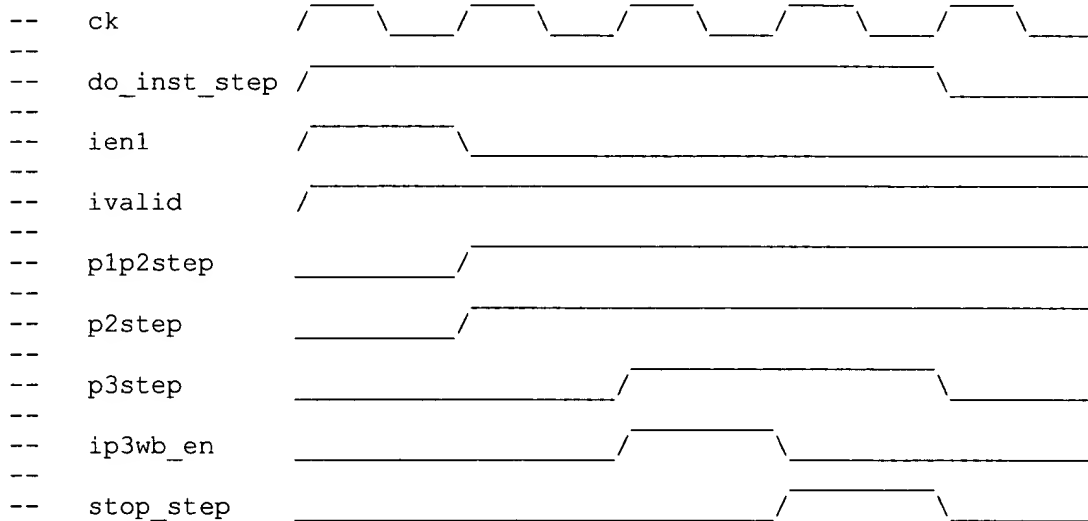
-- The step\_tracker process keeps track on where in the pipeline the  
-- instruction is during single instruction step. It generates three  
-- tracking signals: plp2step, p2step and p3step. The signal p2step  
-- is high when the instruction is in pipestage 2 and p3step is high  
-- when the instruction is in pipestage3. As you see in the timing  
-- diagram below p2step and p3step stays high after being set until  
-- the cycle after the stop signal stop\_step is issued, which means  
-- that the instruction has completed.

--

-- Here is an example how the step tracker process works for an  
-- instruction with writeback and no long immediate. The pipeline  
-- is clean before the step starts.

--

--



-- The signal plp2step is set when a valid instruction has moved from  
-- stage 1 to stage2. This signal sets p2step. But p2step is not only  
-- set by plp2step but also if there is already an instruction in  
-- stage 2 that uses long immediate or has a killed delay slot or if  
-- an interrupt is in stage 2 (p2int is set). This can happen if the  
-- ARC was just halted after running in free-running. The pipeline  
-- can then be filled with anything in this situation. This can only  
-- happen on the first instruction step after free-running mode. On  
-- the second consecutive instruction step the pipeline will be clean.

p2step <= plp2step OR  
          (do\_inst\_step AND (ip2limm OR ip2killnext OR p2int));

step\_tracker: PROCESS(ck, clr)

BEGIN

IF clr = '1' THEN

```

    plp2step <= '0';
    p3step <= '0';
    ELSIF (ck'EVENT AND ck = '1') THEN

        IF istop_step = '1' THEN
            plp2step <= '0';
        ELSIF (ien1 = '1' AND (ivalid = '1' OR plint = '1')) THEN
            plp2step <= do_inst_step;
        END IF;

        IF istop_step = '1' THEN
            p3step <= '0';
        ELSIF ien2 = '1' THEN
            p3step <= p2step;
        END IF;

    END IF;

END PROCESS;

```

```

-----
-- A load of signals inserted to reduce the complexity of the logic
-- minimization task for the ivalid signal

```

```

    ien2_non_iv <= '0' WHEN en = '0'
        OR ien3_non_iv = '0'
        OR (holdup12 OR ihp2_ld_nsc) = '1'
        OR xholdup12 = '1'
        OR ibch_holdp2 = '1' ELSE
        '1';

    ien3_non_iv <= '0' WHEN en = '0'
        OR (xholdup123 AND xt_aluop) = '1'
        OR mwait = '1'
        OR ip3_load_stall = '1'
ELSE
    '1';

```

```

----- ifetch : Tell M/C to do a fetch -----
--
-- This signal is used to tell the memory controller to do another
-- instruction fetch with the program counter value which will appear at
-- the end of the cycle. It is normally the same as pcen except for when
-- the processor is restarted after a reset, when an initial instruction
-- fetch request must be issued to start the ball rolling.
-- In addition, ifetch will be set true when the host is allowed to change
-- the program counter when the ARC is halted. This will means that the new
-- program counter value will be passed out to the memory controller
-- correctly. The ifetch signal is not set true when there is an instruction
-- fetch still pending.
--
-- Signal i_awake will be true for one cycle after the processor is started
-- after a reset.

```

```

    i_awake <= en AND NOT l_go;

```

```

-- Signal i_hostload will be true when an new instruction fetch needs to be

```

```
-- issued due to the host changing the program counter.
```

```
--
i_hostload <= '1' WHEN h_pcwr = '1'
                AND ip2ivalid_r = '1' AND n_go = '1'   else --
                '0';
```

```
i_fetchen <= '0' WHEN en = '0'
--                OR (ip2ivalid_r = '0'
--                and not (i_p2_fst_ifetch_r = '1'
--                and i_fst_ifetch_r = '0'))
--                OR (p2int = '1' AND ien2_non_iv = '0')
--                OR (ip2iv = '1' AND ien2_non_iv = '0')
--                OR (i_break_stage1 = '1')
--                OR inst_stepping = '1'
--                OR plint = '1'                                ELSE
--                '1';
```

```
-- The ifetch signal comes from either pcen, kick-start after reset, or
-- when a fetch is required as the host has changed the program counter.
```

```
--
i_ifetch <= i_fetchen OR i_awake OR i_hostload;
--ARC3 i_ifetch <= pcen OR i_awake OR i_hostload;
```

```
-- The latch is set true after the processor is started after a reset, and
-- will stay true until the next reset.
```

```
--
-- l_go is taken low when the instruction cache is invalidated
-- This is in order to prevent a lockup situation
```

```
n_go <= en OR l_go;
ni_go <= n_go AND not ivic;
```

```
lego: PROCESS(ck, clr)
```

```
BEGIN
```

```
IF clr = '1' THEN
    l_go <= '0';
ELSIF (ck'EVENT AND ck = '1') THEN
    l_go <= ni_go;
END IF;
```

```
END PROCESS;
```

```
----- ipending : An instruction is being fetched -----
```

```
--
-- This signal is set true when an instruction fetched has been issued,
-- (i.e. not directly after reset) and the fetch has not yet completed,
-- signaled by ivalid = '0'.
-- It is used to prevent writes to the pc from the host from generating
-- an ifetch request when there is already an instruction fetch pending.
-- Host accesses are rejected with hold_host, generated in hostif.vhd
--
```



```

ipend : process(ck, clr)
begin
  IF clr = '1' THEN i_ipending <= '0';
  ELSIF ck = '1' AND ck'event THEN

    -- entry state : when ARC is started onwards
    -- IF i_ifetch = '1' THEN i_ipending <= '1';
    -- END IF;

    -- entry state : when ARC is started onwards --
    IF i_ifetch = '1' THEN i_ipending <= '1';
    END IF;

    -- exit state : i.e. when no more fetches are required
    --
    -- Or: An instruction cache invalidate puts us back into
    -- the immediately post-reset condition.
    --
    IF (i_ifetch = '0' AND invalid = '1')
    OR (ivc = '1') THEN ipending <= '0';
    END IF;

    --Pending ifetchs in the core are cancelled when an invalid
    --returns from the cache or an invalidate is requested
    --
    IF (--i_ifetch = '0' AND
        invalid = '1')
    OR (ivc = '1') THEN i_ipending <= '0';
    END IF;

  END IF;
END PROCESS;

----- en1 : Pipeline 1 -> 2 transition enable -----
--
-- This signal is true at all times when the processor is running except
-- when:
--
-- a. A valid instruction in stage 2 cannot complete for some reason, or
-- if an interrupt in stage 2 is waiting for a pending instruction fetch
-- to complete.
--
-- b. A breakpoint instruction (or valid actionpoint) is detected and
-- stage 2 has to be halted, while the remaining stages are flushed, and
-- then halted.
--
-- c. The single instruction has already moved on to stage 2 and this
-- instruction does not depend on the following instruction.
-- This is a special case that only happens during single instruction
-- step. Because single instruction step finishes the instruction that
-- was in pipeline stage 1, this is actually the starting mechanism of the
-- single instruction stepping. The next instruction is not allowed
-- to pass on until the instruction in further down the pipe has
-- completed and not until a new single instruction step command

```

```
-- has been generated.
--
-- *** Note that if an invalid instruction in stage 2 is held (this will be
-- due to a stall at stage 3) then the instruction in stage 1 will be
-- allowed to move into stage 2. ***
--
```

```
--An additional disable flag is added to cope with ifetch stalling and
--the cache keeping ivalid high even though the instruction in stage 1
--has moved to stage 2 or further.
--
```

```
    ien1    <= '0' WHEN en = '0'
              OR (p2step = '1' AND ip2pldep = '0' AND p2int = '0')
              OR (i_break_stagel = '1')
              OR (p2int = '1' AND ien2 = '0')
              OR (ip2iv = '1' AND ien2 = '0')
              OR i_pl_used_r = '1'
                -- or (i_ifetch_r = '0' and i_ipending = '0')
                -- or (i_ipending = '0' and i_awake = '0')
              ELSE
                '1';
```

```
-- The signal ien1_lowpower (below) is almost always equal to ien1 (above),
-- except
-- when the opcode is not valid. This prevents invalid opcodes to propagate to
-- pipeline stage 2 and thereby power is saved.
--
-- This is ONLY used to enable the p2iw latch. The global EN1 stays as normal.
-- The ivalid signal is also used in sync_regs to switch off RAM reads when the
-- new instruction is not valid.
--
```

```
    ien1_lowpower <= '0'    WHEN (ivalid = '0')    ELSE
                      ien1;
```

```
----- en2 : Pipeline 2 -> 3 transition enable -----
```

```
-- This signal is true when the processor is running, and the instruction
-- in stage 2 can be allowed to move on into stage 3. It may be held up for
-- a number of reasons:
```

- a. A register referenced by the instruction is currently the subject
-- of a pending delayed load. (holdup12 from the scoreboard unit).
- b. Stage 2 contains an instruction which requires a long immediate
-- data value from stage 1 which cannot be fetched on this cycle.
-- (ip2limm = '1')
- c. Stage 2 contains a jump/branch instruction, which require that the
-- correct instruction be present in the delay slot following the
-- jump/branch instruction.
-- ( ip2bch = '1', ivalid = '0')
- d. An interrupt in stage 2 is waiting for a pending instruction
-- fetch to complete before issuing the fetch from the interrupt
-- vector.
- e. A valid instruction in stage 3 is held up for some reason.

```

--      - Note that stage 3 will never be held up if it does not contain
--      a valid instruction.
--
--      f. Extensions require that stage 2 be held up, probably due to a
--      register not being available for a read on this cycle.
--
--      g. The branch protection system detects that an instruction setting
--      flags is in stage 3, and a dependent branch is in stage 2. Stage 2
--      is held until the instruction in stage 3 has completed.
--
--      h. The opcode is not valid (ip2iv = '0') and this is not due to an
--      interrupt (i.e p2int = '0'). This is done to reduce power
consumption.
--
--      i. The actionpoint debug mechanism or the breakpoint instruction
--      is triggered and thus disables the instructions from
--      going into stage 3 when the instruction in stage 1 is the delay slot
--      of a branch/jump instruction.
--
--      j. A branch/jump with a delay slot that is not killed is in stage 2
--      during single instruction step.
--
-- All ivalid have been changed to p2ivalid_r so the processor doesn't
-- get more than one cycle ahead
-- Additionally instructions in stage 2 referencing a limm can only move
-- after the limm is merged.
-- Also when stage 2 is stalled when p1 has be used ...

```

```

--      ien2      <= '0' WHEN en = '0'
--                  OR ien3 = '0'
--                  OR (holdup12 OR ihp2_ld_nsc) = '1'
--                  OR xholdup12 = '1'
--                                     or (i_p2merge_valid_r = '0' and
--                                     p2limm = '1')
--
--                  OR (p2int = '1' AND ip2ivalid_r = '0')
--                  OR (ip2bch = '1' AND ip2ivalid_r = '0')
--                  OR (ip2limm = '1' AND ip2ivalid_r = '0')
--                  OR ibch_holdp2 = '1'
--                  OR (ip2iv = '0' AND p2int = '0')
--                  OR (i_break_stage2 = '1')
--                  or i_pl_used_r = '1'
--                  OR (plp2step = '1' AND ip2bch = '1'
--                  AND ip2limm = '0' AND ip2killnext = '0')
--                                     ELSE
--
--                  '1';

```

```

----- ibch_holdp2 : Branch protection system -----

```

```

-- In order to reduce code size, we want to remove the need to have a NOP
-- between setting the flags and taking the associated branch.

```

```

-- e.g.          sub.f      0,r0,23          ; is r0=23?
--              nop                ; padding instruction. <-----
--              bz          r0_is_23        ;

```

```
-- In order that the compiler does not have to generate these instructions,
-- we can generate a stage 2 stall if an instruction in stage 3 is attempting
-- to set the flags. Once this instruction has completed, and has passed out
-- of stage 3, then stage 2 will continue.
```

```
-- We need to detect the following types of valid instruction at stage 3:
```

- ```
-- i. Any ALU instruction which sets the flags (p3setflags)
-- ii. Jcc.F or JLcc.F
-- iii. A FLAG instruction.
```

```
ibch_p3flagset <= ip3iv WHEN (ip3setflags = '1') --
ALU
OR ((ip3i = ojcc) AND (ip3_fbit = '1')) --
Jcc/JLcc
OR ((ip3i = oflag) AND (ip3c = so_flag)) ELSE --
FLAG
'0';
```

```
-- In order to generate the stall, we also need to detect a valid branch
instruction
```

```
-- present in stage 2 (ip2bch).
```

```
-- We generate a stall when the two conditions are present together:
```

- ```
-- a. An instruction in stage 3 is attempting to set the flags
-- b. A branch instruction at stage 2 needs to use these new flags
```

```
-- Note that it would be possible to detect the following conditions to give
-- theoretical improvements in performance. These are very marginal, and have
-- been left out here for the sake of simplicity, and the fact it would be
-- difficult for the compiler to take advantage of these optimizations.
```

```
-- Both cases remove the link between setting the flags and the following
-- branch, either because the flags don't get set, or because the branch doesn't
-- check the flags.
```

- ```
-- i. Conditional flag set instruction at stage 3 does not set flags
-- e.g. add.cc.f r0,r0,r0, resulting in C=1
```

- ```
-- ii. Branch at stage 2 uses the AL (always) condition code.
```

```
ibch_holdp2 <= '1' WHEN (ibch_p3flagset = '1') -- p3 setting
flags
AND (ip2bch = '1') ELSE -- branch in p2
'0';
```

```
----- en3 : Stage 3 instruction completion control -----
```

```
-- This signal is true when the processor is running, and the instruction
-- in stage 3 can be allowed complete and set the flags if appropriate.
-- Stage 3 may be prevented from completing for a number of reasons:
```

- ```
-- a. An extension multi-cycle ALU operation has requested extra time
-- to complete the operation (xholdopl23). Note that this can only
-- be the case when extension alu operations are enabled with the
```

```

--      xt_aluop constant in extutil.vhd.
--
--      b. The memory controller is busy and cannot accept any more load or
--      store operations. (mwait)
--
--      c. Deleted in v6.
--
----- p2iv : Stage 2 instruction valid -----
--
-- This signal indicates that stage 2 contains a valid instruction. The
-- instruction in stage 2 may not be valid for a number of reasons:
--
--      a. A breakpoint/actionpoint has been detected, and instructions in
--      stage two are to be invalidated for when the ARC is to be
--      restarted.
--
--      b. The correct instruction word could not be fetched in time, so
--      a junk instruction is inserted into the pipeline to keep it
--      flowing.
--
--      c. An interrupt was recognized, causing the instruction which was in
--      stage 1 (valid or not) to be killed.
--
--      d. The interrupt which was recognized, and which is now in stage 2,
--      requires a blank delay slot to perform the jump to the interrupt
--      vector. The instruction in stage 1 is therefore killed.
--
--      e. A long immediate data value was required by the previous
--      instruction, and is killed to prevent it being executed as a
--      real instruction.
--
--      f. The delay slot mechanism of a jump/branch instruction in stage 2
--      has decided that the following instruction should be killed.
--      Note that this instruction must be present in stage 1 in order
--      to be killed, before the pipeline can be moved on. This is
--      handled by the en2 signal.
--
--      g. The single instruction in stage 2 will move on to stage 3 the
--      next cycle. This is a special case which only occurs during
--      a single instruction step. This must be done to avoid the
--      instruction from being executed repeatedly in stage 2. The
--      reason this does not kill instructions with long immediates
--      or delay slots is because of the signal ien2. The signal ien2
--      is not set when there is an instruction in stage 2 that uses a
--      long immediate or delay slot in stage 1 in this situation. The
--      reason is that stage 2 stalls while another fetch is being done
--      in order to get the L IMM/delay slot.
--
-- The appropriate value is latched into p2iv when the instruction in stage 1
-- is allowed to move into stage 2.
--
--Jumps can move independently of delay slot instructions in this case
-- delays which need to be killed are killed by pending kill logic
(i_pending_kill_r)
  n_p2iv <= '0'      WHEN ((i_break_stage1 = '1') AND
                           (i_break_stage2 = '0') AND ien2 = '1') OR

```

```

                (p2step = '1' AND ien2 = '1')          ELSE
ip2iv          WHEN ien1 = '0'                        ELSE
'0'            WHEN (plint OR p2int) = '1'
                OR ip2limm = '1'
                OR ip2killnext = '1'                    ELSE

                OR ip2killnext = '1'
                or i_pending_kill_r = '1'

ELSE
    ivalid;

p2ivreg :      PROCESS(ck, clr)

    BEGIN

    IF clr = '1' THEN
        ip2iv <= '0';
    ELSIF (ck'EVENT AND ck = '1') THEN
        ip2iv <= n_p2iv;
    END IF;

END PROCESS;

----- p3iv : Stage 3 instruction valid -----
--
-- This signal indicates that stage 3 contains a valid instruction. The
-- instruction in stage 3 may not be valid for a number of reasons:
--
--    a. The instruction was marked as invalid when it moved into stage 2,
--       i.e. p2iv = '0'.
--
--    b. The instruction in stage 2 has not been able to complete for some
--       reason, and the instruction in stage 3 has been able to complete
--       and will move on at the end of the cycle. It is thus necessary
--       to insert a blank slot into stage 3 to fill in the gap. If this
--       is not done, the instruction which was in stage 3 will be
--       executed again, and this would of course be *bad news*.
--
n_p3iv <= ip3iv    WHEN ien3 = '0'                      ELSE
'0'          WHEN ien2 = '0' AND ien3 = '1'            ELSE
ip2iv;

p3ivreg :      PROCESS(ck, clr)

    BEGIN

    IF clr = '1' THEN
        ip3iv <= '0';
    ELSIF (ck'EVENT AND ck = '1') THEN
        ip3iv <= n_p3iv;
    END IF;

END PROCESS;

```

```

----- Disable Logic to Stall the ARC -----
----- for Actionpoint System -----

```

```

--
-- The pipeline flushing mechanism has been introduced to support the
-- breakpoint instruction and actionpoint hardware. Each stage of the
-- pipeline is stalled explicitly, and once all stages one, two and three
-- have been stalled the ARC is stalled via en bit
--

```

```

-- This signal is true when both of the the following conditions are true:
--

```

- a. The instruction in stage one should be killed when it advances
-- into stage two.
- b. The actionpoint mechanism was set by the hardware breakpoint
-- alone.

```

i_kill_AP      <= ip2killnext AND hw_brk_only;

```

```

-- The stalling signal for stalling en1 is defined by i_break_stagel, and
-- this is set to '1' on the following conditions:
--

```

- a. The breakpoint instruction has been detected at stage one, i.e.
-- i\_brk\_inst = '1' or an actionpoint has been triggered by a valid
-- signal from the OR-plane.
- b. The instruction in stage one of the pipeline is to be executed,
-- and not killed.
- c. The sleep instruction has been detected in stage 2.
- d. The ARC is sleeping already (sleeping = '1') due to a sleep
-- instruction that was encountered earlier.

```

i_break_stagel  <= '1' WHEN  i_brk_inst = '1'
                        OR  ip2sleep_inst = '1'
                        OR  sleeping = '1'
                        OR  (actionhalt = '1'
                        AND i_kill_AP = '0')
                        ELSE
                        '0';

```

```

-- The stalling signal for stalling en2 is defined by i_break_stage2, and
-- this is set to '1' on the following conditions:
--

```

- a. A breakpoint/actionpoint instruction has been detected at stage
-- one, i.e. i\_brk\_inst = '1'. For example, an actionpoint has
-- been triggered by a valid signal from the OR-plane,
-- This has to true when there is an instruction
-- in stage one is in the delay slot of a branch, jump or loop
-- instruction. It can also be long immediate data.
- b. A breakpoint/actionpoint instruction has been detected at stage
-- one, i.e. i\_break\_stagel = '1'. For example, an actionpoint has
-- been triggered by a valid signal from the OR-plane.
-- This has to true when there is an interrupt in
-- stage two, i.e. p2int = '1'

```

--
i_break_stage2 <= '1'      WHEN ((ip2pldep = '1' OR p2int = '1')
                                AND (i_break_stagel = '1'))      ELSE
                                '0';

-- As the pipeline is flushed of instructions when the breakpoint instruction
-- or a valid actionpoint is detected it is important to disable each stage
-- explicitly. These signals have to follow the last instruction which is being
-- allowed to complete. A normal instruction in stage one will mean that
-- instructions in stage two, three and four will be allowed to complete.
However,
-- for an instruction in stage one which is in the delay slot of a branch, loop
-- or jump instruction means that stage two has to be stalled as well.
Therefore,
-- only stages three and four will be allowed to complete.
--
-- The qualifying valid signal for stage two is defined by i_n_AP_p2disable,
-- and this is set to '1' on the following conditions:
--
--     a. There is an instruction in stage two which has a dependency in
--         stage one, i.e. i_break_stage2 = '1'.
--
--     b. The breakpoint instruction or actionpoint has been detected, i.e.
--         i_break_stagel = '1' and the instruction in stage two is enabled,
--         ien2 = '1', and the instruction is allowed to move on.
--
--     c. The breakpoint instruction or actionpoint has been detected, i.e.
--         i_break_stagel = '1' and the instruction in stage two is invalid,
--         ip2iv = '0'.
--
i_n_AP_p2disable <= '1'      WHEN i_break_stage2 = '1' OR
                                (i_break_stagel = '1' AND
                                 ((ien2 = '1' AND ip2iv = '1') OR
                                  ip2iv = '0'))
                                ELSE
                                '0';

-- The qualifying valid signal for stage three is defined by i_n_AP_p3disable,
-- and this is set to '1' on the following conditions:
--
--     a. The instruction in stage two is invalid, i_AP_p2disable_r = '1'.
--         Also the instruction in stage three is enabled, en3 = '1', and
--         the instruction is allowed to move on.
--
--     b. The instruction in stage two is invalid, i_AP_p2disable_r = '1'.
--         Also the instruction in stage three is invalid, ip3iv = '0'.
--
i_n_AP_p3disable <= '0'      WHEN i_break_stagel = '0'              ELSE
                                '1'      WHEN (i_AP_p2disable_r = '1') AND
                                ((ien3 = '1' AND ip3iv = '1') OR
                                 ip3iv = '0')              ELSE
                                '0';

update_AP_disable :    PROCESS(ck, clr)

BEGIN

```



```

IF clr = '1' THEN
    i_AP_p2disable_r <= '0';
    i_AP_p3disable_r <= '0';
ELSIF (ck'EVENT AND ck = '1') THEN
    i_AP_p2disable_r <= i_n_AP_p2disable;
    i_AP_p3disable_r <= i_n_AP_p3disable;
END IF;

END PROCESS;

-- Output Dives for halting the ARC

AP_p3disable_r <= i_AP_p3disable_r;

END synthesis;

```